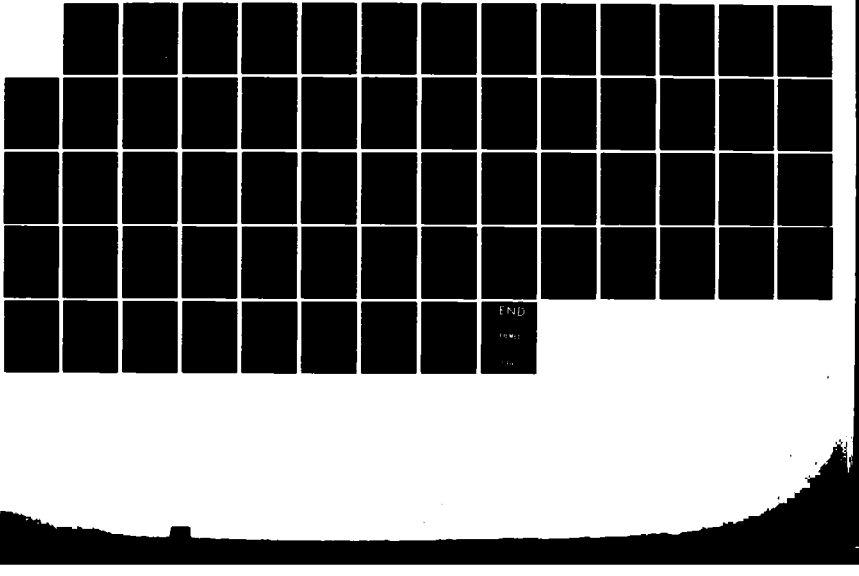
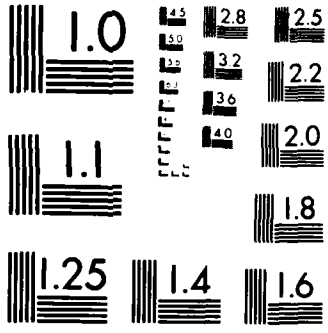
AD-A149 783 CONCURRENT SIMULATION TECHNIQUES EXPLOITING HIERARCHY (U) ILLINOIS UNIV AT URBANA COMPUTER SYSTEMS GROUP W A ROGERS AUG 84 CSG-34 1/1

UNCLASSIFIED F/G 9/5 NL





(2)

REPORT CSG-34 AUGUST, 1984

CONCURRENT SIMULATION TECHNIQUES EXPLOITING HIERARCHY

WILLIAM ARTHUR ROGERS

UNI

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CSG #34	5. MONITORING ORGANIZATION REPORT NUMBER(S) N/A
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Laboratory University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION Semiconductor Research Corporation
6c. ADDRESS (City, State and ZIP Code) 1101 West Springfield Avenue Urbana, IL 61801	7b. ADDRESS (City, State and ZIP Code) 300 Park Drive, Suite 215 P.O. Box 12053 Research Triangle Park, NC 27709
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Semiconductor Research Corp. 8b. OFFICE SYMBOL (If applicable) N/A	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER SRC RSCH 83-01-014
8c. ADDRESS (City, State and ZIP Code) 300 Park Drive, Suite 215 P.O. Box 12053 Research Triangle Park, NC 27709	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO.: N/A PROJECT NO.: N/A TASK NO.: N/A WORK UNIT NO.: N/A
11. TITLE (Include Security Classification) Concurrent Simulation Techniques Exploiting Hierarchy	
12. PERSONAL AUTHOR(S) Rogers, William Arthur	
13a. TYPE OF REPORT Technical 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day) August 1984 15. PAGE COUNT 57
16. SUPPLEMENTARY NOTATION N/A	
17. COSATI CODES FIELD / GROUP / SUB. GR.	18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) hierarchical fault simulation, fault library, SCALD, macromodule, concurrent fault simulation, deductive fault simulation

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Current fault simulation techniques such as concurrent, deductive, and parallel fault simulation are not powerful enough for today's very large integrated circuit designs. More powerful fault simulation techniques are needed to prevent a crisis in integrated circuit testing. A new simulation technique, based on the well-known concurrent and deductive techniques, is presented which uses a hierarchical representation of the circuit design and, unlike the traditional implementations of these techniques, does not expand the circuit to a single, lowest level, description. The simulation technique is shown to be decoupled from the fault model of the circuit through the use of fault libraries. These libraries are based on the principle that any detectable fault will cause an erroneous output value for some input vector. The implementation of this technique is described and preliminary performance results are given. The advantages and disadvantages of this technique are discussed and possible enhancements are described.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code) 22c. OFFICE SYMBOL N/A

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

CONCURRENT SIMULATION TECHNIQUES EXPLOITING HIERARCHY

BY

WILLIAM ARTHUR ROGERS

B.S., Tulane University, 1980

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1984

Urbana, Illinois

ABSTRACT

Current fault simulation techniques such as concurrent, deductive, and parallel fault simulation are not powerful enough for today's very large integrated circuit designs. More powerful fault simulation techniques are needed to prevent a crisis in integrated circuit testing. A new simulation technique based on the well-known concurrent and deductive techniques is presented, which uses a hierarchical representation of the circuit design and unlike the traditional implementations of these techniques does not expand the circuit to a single, lowest level, description. The simulation technique is shown to be decoupled from the fault model of the circuit through the use of fault libraries. These libraries are based on the principle that any detectable fault will cause an erroneous output value for some input vector. The implementation of this technique is described and preliminary performance results are given. The advantages and disadvantages of this technique are discussed and possible enhancements are described.

ACKNOWLEDGMENT

The author wishes to thank Professor Jacob A. Abraham for advising the author during this research. Professor Abraham's encouragement, insight, and enthusiasm were invaluable contributions to this work. His continuous stream of new ideas was very stimulating.

The author wishes to thank Professors Edward S. Davidson, Janak H. Patel, and all his colleagues in the Computer Systems Group at the Coordinated Science Laboratory for their friendship and intellectual stimulation.

The author would also like to thank Hewlett-Packard Company and the Semiconductor Research Corperation (contract RSCH 83-01-014) for their support, which helped make this research possible.

Finally, the author would like to thank his wife and parents for their continual love, support, and encouragement.

TABLE OF CONTENTS

1. INTRODUCTION

Simulation is the technique of approximating the response of a system to a stimulus by evaluating a model of the system. The accuracy of the model determines how closely the simulation approximates the actual system and greater accuracy is usually more expensive in terms of computer time and space. When the computation time becomes unacceptably long, or the space requirements exceed the capabilities of the computer, simulation becomes impractical. The alternatives to this situation are: use a larger, faster computer; reduce the computational requirements by simplifying the models; break the system into smaller pieces and simulate each piece separately; or use more efficient algorithms.

The first approach, using larger, faster computers, is practical in some instances, but many simulation algorithms show second order effects (or worse) [1,2] in time and space so a machine four times "bigger" would be required to simulate a system only twice as large. Clearly this is an expensive solution, and many problems exist which cannot be simulated on any machine in existence today.

The second approach, simplifying the models, has been successful, but it involves trading accuracy for speed and space. A typical example of this method is using logic simulation instead of circuit simulation to simulate large, integrated circuit designs. Circuit simulators can handle on the order of 100 transistors [3], while logic simulation can

handle several thousand transistor circuits. The tradeoff is reduced accuracy of signal values, and in some designs may yield values that are wrong. The user of a simulator must be aware of the limitations imposed by simulation models and stay well within these limitations to avoid incorrect results. Since very few physical systems can be modelled perfectly, this is a fundamental problem in simulation, and the user should always check the results of a simulation for validity. The accuracy versus speed and space problem has led to the development of multi-mode simulation where circuit simulation (most expensive) is used for critical timing paths, logic simulation (less expensive) is used for other circuitry directly interacting with the critical path, and functional simulation (least expensive) for the remaining parts of the system [4].

The third approach, partitioning the circuit, can be difficult and tedious for the user [5,6,7]. In cases where the simulation algorithm is at least $O(n^2)$, partitioning may dramatically increase throughput by reducing the number of primitives the algorithm deals with at any one time. That is, a large N^2 is much greater than the sum of its squared factors. Even for a linear algorithm, if the circuit is so large that frequent page faults seriously degrade the performance of the simulator, partitioning can improve throughput by reducing the page fault rate, but the total amount of work required to evaluate the circuit primitives remains unchanged. Manual partitioning is a poor means of fitting the circuit to the simulator, because it is tedious and error prone.

The final approach, using more efficient algorithms, can be viewed as the evolution of a particular style of simulation. For example, the earliest fault simulations were done with logic simulators and the faults were injected by manually altering the system being tested [8]. Fault simulation quickly evolved to specialized simulators that automatically injected the faults, and then to more efficient algorithms for performing this simulation. This evolution led to the parallel simulator which provided tremendous speedup by simulating several closely related machines at the same time. Parallel simulation is still a popular simulation technique. From parallel simulation, deductive [9] and concurrent [10] simulation were the next major developments in fault simulation algorithms. These techniques are fundamentally different from parallel simulation because they determine all detectable faults in the system, for a particular input vector, in one pass instead of many passes. These techniques have been the subject of current research and development in fault simulation and are becoming quite popular in industry. Experience has shown that these techniques are more efficient on large systems than parallel simulation, but they require much more memory [11]. Since the price of computer memory is decreasing rapidly, concurrent and deductive fault simulation techniques are steadily becoming more attractive.

1.1. Terminology

A failure is a defect which if present in the circuit may, under the appropriate conditions, cause the circuit to behave incorrectly.

The description of the effect of this failure at some level of abstraction is called a fault. The incorrect behavior of the fault is called an error. The term fault is often used interchangeably in the literature where error is really meant. Errors are said to propagate (through a module) when the error presented at the input to the module causes an incorrect response (error) at the outputs of the module. If no incorrect response is generated, then the error is said to be absorbed or blocked. If, for the input stimulus and a particular fault, no error is produced at the primary outputs, then that fault is undetectable under the current input. In the case of sequential circuits, the current and all previous inputs must be considered. If the fault remains undetectable for all possible input stimuli, then the fault is an undetectable fault.

There are two intrinsic tasks which a fault simulator must perform: fault activation and fault propagation. Activation is the process of deciding which internal faults can affect the outputs of a module, given the input stimulus, and propagation is the process of deciding if the faults present at the inputs of a module can be detected at the outputs of the module. The activation process used by the author is based on a table lookup technique and is described in Chapter 3.

1.2. Research Goals

We believe that the use of hierarchy wherever possible is important to improve simulator performance in several ways. The hierarchical representation of a system is more compact, which helps increase the

size of the system that can be simulated. This hierarchical representation also allows the user to control the complexity of the simulation by controlling the complexity of the system description [12]. The hierarchical description of a system also affords different perspectives of the system which are difficult to obtain from a flat system description. These perspectives are invaluable for tools that require knowledge of the system structure, such as fault diagnosis and test generation tools [13]. There appears to be a trend toward hierarchical representations of systems for computer-aided design (CAD) tools because using hierarchy is a natural technique for reducing the apparent complexity of a system by providing abstractions of each level of the system [5,14,15]. In order to fully utilize the power of hierarchical representation the tools must use the hierarchy internally, rather than flattening a hierarchical description, and perform all reporting to the user in terms of the hierarchy. For these compelling reasons we believe it is important to orient our work towards hierarchical representations.

The goals of this research were to develop an alternative fault simulation technique that was based on hierarchical system evaluation. We also wanted our technique to incorporate the advantageous features of both concurrent and deductive simulation, but without many of the disadvantages. The hybrid technique which is detailed in Chapter 3 is inherently hierarchical and uses an unordered list-based propagation technique. This avoids the ordered list operations and deduction equations of the deductive technique and also avoids (when possible) the replication of machines which penalizes the concurrent technique.

Most fault simulators are tied to a particular technology through the fault model embedded in the simulator. Changing the fault model usually requires modifying the simulator, a task few users want to perform. Our final goal then was to increase the usefulness of our fault simulation technique by decoupling the fault model from the simulator, so that a change in fault models requires changing data files, not modifying programs.

2. COMPARISON OF FAULT SIMULATION TECHNIQUES

An informal comparison of simulation techniques quickly shows why concurrent simulation is more efficient than parallel simulation. It also shows why parallel simulation performs better for smaller circuits, and concurrent simulation for larger circuits. For any type of fault simulator, each fault must be examined and its consequences applied to the circuit. Since the circuit is represented as a collection of simulator primitives, the effect of these faults must be determined from evaluating the primitives, and similarly the effect of the faults on other parts of the system must be determined by applying the result of the faults to these other elements to determine their response.

If the various simulation techniques can be modelled by equations derived from characteristics of the simulation algorithms, then these equations can be used to predict simulator performance. In this chapter simple performance models of parallel, deductive, and concurrent fault simulation are derived from their algorithms. These models are used to help explain the performance characteristics of the various techniques. Finally, the models are used to hypothesize approaches to improving fault simulator performance. To simplify the models, we neglect any one-time overhead such as initialization or output which is relatively independent of the circuit size or composition and assume measured averages for terms that vary with circuit topology or input stimulus. The

cost factor derived here is called the primitive fault product or PFP and is a function of the following parameters:

F = total number of possible faults for the circuit
f = average number of faults for each primitive (parallel)
f' = average number of faults for each primitive (deductive)
f" = average number of faults for each primitive (concurrent)
w = word width of the target computer
b = number of bits used to represent a logical value
P = number of circuit elements (simulator primitives)
K = (w/b) - 1 (faulty machines evaluated in parallel)
a = activity factor in event-driven simulation ($0<a<<1$)
b = fault collapsing factor ($0<b<1$)
c = cost per evaluation (cpu seconds)

Since changes in input may not affect many of the signals within a system being simulated, event-driven simulation improves performance by evaluating only the signals that change (the activity). Typically, this activity affects only 5-20% of the system, so 80% or more of the evaluation performed by a compiled simulator is avoided. The activity factor a accounts for this improvement. The fault collapsing factor b accounts for the reduction in the number of faults the simulator deals with due to fault folding, fault collapsing, and fault dominance. Evaluations for the cost factor c are the number of then most meaningful simulator events for each technique. For parallel simulation, an evaluation is the processing of one primitive for a group of machines. In deductive and concurrent simulation, an evaluation is the processing of one fault or list of equivalent faults.

2.1. Parallel Fault Simulation

In parallel fault simulation, word instructions are used to evaluate the same device in several different faulty machines at the same time. The virtual data structure, shown in Figure 1, demonstrates how this is accomplished. The columns represent the nodes of the circuit and the rows represent the different faulty machines. Row 0 represents the fault-free circuit. Because the nodes are aligned with the words, and the faulty machines across the words, a computer instruction, performing a logical operation on two words, simultaneously performs the evaluation of that operation on the same nodes for several different machines. This parallelism significantly speeds up the fault simulation. The speedup factor depends on how many machines can be packed into one word. A more detailed explanation of the parallel fault simu-

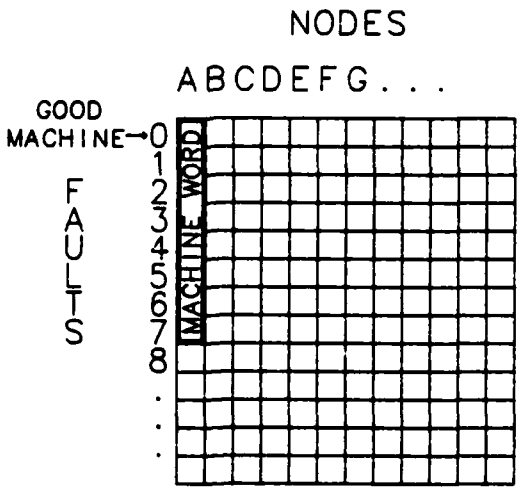


Figure 1. Virtual Data Structure for Parallel Simulation

lation algorithm can be found in [8].

The virtual data structure previously described can also be used to develop a measure of the computation performed during a fault simulation. The evaluation of all faulty machines requires either F/K or F/K+1 passes. Since each primitive must be evaluated at least once per pass, in a compiled simulation (more if the circuit contains feedback loops), the total number of evaluations performed is $P*(F/K)$. Substituting fP for F shows the second order nature of the parallel fault simulation. There are several techniques for increasing the speed of parallel simulation, such as fault folding and activity directed (event-driven) simulation, but none of these techniques changes the order of the algorithm. The expanded form of this equation then is

$$PFP = abcP(fP)/((w/b)-1)$$

2.2. Deductive Fault Simulation

Deductive fault simulation, developed by Armstrong [9] simulates only the good machine and computes the effect of the faults with fault list equations. These equations, defined here as deduction equations, perform the operations of set intersection, union, and complement on fault lists and the super fault list. The super fault list is the list of all possible faults and the complement of a fault list is defined as all faults not present in the list, or the super fault list minus the list to be complemented. The set intersection and union computations become slow when they involve large sets because these operations

perform insertions and deletions on ordered sets. The complement operation is especially costly because it involves copying (with deletions) the super fault list.

The deduction equations are data dependent and must be derived during simulation. For a gate with all inputs at non-controlling values (0 for an OR and 1 for an AND) the deduction equation is derived by the formula: The output list is the union of all the input lists and the output stuck at the controlling value.

$$\left\{\begin{matrix}\text{output}\\ \text{list}\end{matrix}\right\} = \left\{\begin{matrix}\text{union of}\\ \text{all input}\\ \text{lists}\end{matrix}\right\} \cup \left\{\begin{matrix}\text{output stuck}\\ \text{at controlling}\\ \text{value}\end{matrix}\right\}$$

If some of the inputs are at controlling values (1 for an OR and 0 for an AND) then the equation becomes more complicated. The output list is the output stuck at the non-controlling value and the intersection of the complement of the union of non-controlling input lists and the union of controlling input lists.

$$\left\{\begin{matrix}\text{output}\\ \text{list}\end{matrix}\right\} = \left\{\begin{matrix}\text{output stuck at}\\ \text{non-controlling}\\ \text{value}\end{matrix}\right\} \cup \left[\overline{\left\{\begin{matrix}\text{union of}\\ \text{non-controlling}\\ \text{input lists}\end{matrix}\right\}} \cap \left\{\begin{matrix}\text{union of}\\ \text{controlling}\\ \text{input lists}\end{matrix}\right\}\right]$$

More details of this technique can be found in Baker [10]. The deductive technique is not as versatile as other fault simulation techniques because of its list-based algorithm, but recent and advances in string processors may dramatically change this situation very soon [16]. More recent work has generalized the deductive simulation technique [17]. A comparison of several fault simulation techniques is presented in Levendel [18] which is particularly interesting because it considers extended

versions of each technique.

In deductive simulation the number of faults evaluated at each primitive is the sum of the number of faults present at the inputs of the primitive, the output fault, and the total number of faults F if any lists are complemented. The number of faults present on the inputs is much less than the total number of faults so the frequency of list complement operations strongly influences the average number of faults processed at each primitive f'. The cost for performing each pass of a deductive simulation is:

$$PFP = abcPf'$$

2.3. Concurrent Fault Simulation

The concurrent fault simulation algorithm was developed by Ulrich and Baker [10] and is characterized by scheduling the good machine and all faulty machines in the same event queue. More recent work has been reported which emphasizes improving the performance aspects of concurrent simulation [19]. In this technique simulation begins by applying a vector to the primary inputs and evaluating the first available primitive for the good machine. As each primitive is evaluated, the faults in that primitive that are activated by the current stimulus spawn new machines. These machines differ from the good machine by the effect of the fault. There is no difference in the processing of the faulty machines and the good machine except that only the good machine triggers the spawning of new machines. Spawning more faulty machines

from faulty machines would effect a multiple fault evaluation. As each faulty machine is spawned, the current state of the good machine is duplicated, and the new machine is added to the evaluation queue. Clearly, duplicating the entire state of the good machine for each faulty machine produces maximum flexibility, but is very expensive in terms of memory space.

Concurrent fault simulation is inherently an event-driven algorithm. Therefore the PFP is dependent on the input stimulus, which determines what faults are activated, propagated, or absorbed. The total number of evaluations is the summation over the primitives, of all the faults propagated to each primitive, plus all the faults activated in that primitive, plus the good machine evaluation. This metric, because of its sensitivity to the input, is difficult to use, instead, use the average number of faults evaluated per primitive f", a measurable quantity. Since f" is generally large, any technique which decreases the average number of faults evaluated per primitive will significantly improve the performance of the simulator. A dynamic fault collapsing technique, detailed in Chapter 3, decreases this average to the minimum possible value.

The PFP then is the sum of good machine evaluations P, faulty machine evaluations Pf", and the appropriate constants for activity and cost:

$$PFP = acP(f''+1)$$

This equation appears linear, but f" is dependent on the number of primitives, the input, and circuit topology.

2.4. Characteristics of the Simulation Models

Although these models are quite simple they highlight the salient parameters that affect simulator performance. The models show that the performance of all the techniques is a function of the product of primitives and faults, which implies that performance can be improved by reducing this product. The number of faults can be reduced by partitioning the fault set or static fault collapsing, and the number of primitives can be reduced by redefining the system in terms of more comprehensive primitives. There is a caveat in the last approach because more complex primitives encompass more faults so the average number of faults per primitive increases with primitive complexity and the primitive-fault product may not change very much. However, fault reduction techniques are more successful with the more complicated "primitives", so the total number of faults and therefore the primitive-fault product can be reduced.

2.5. Concurrent Versus Deductive Simulation

The cost factor for parallel simulation is much smaller than the cost per evaluation in concurrent simulation because the parallel evaluation is much less complicated, involves less overhead manipulating data structures, and is amortized over several different machines. For small numbers of faults and primitives, parallel simulation is faster,

but for large numbers of faults and primitives concurrent simulation is faster. The crossover point seems to be around 1000 simulator primitives [11].

In this section the two most viable techniques for fault simulation are compared in sufficient detail to show the advantages and disadvantages of each. The discussion is intended to motivate the development of a hybrid algorithm which incorporates the advantages of both concurrent and deductive simulation.

The main disadvantages of deductive fault simulation are that it requires storage of long, ordered fault lists at each node, and the processing of these ordered lists is expensive, particularly the complement operation. Since the complement operation occurs very frequently, it severely degrades simulator performance. This technique also suffers a large memory penalty because the super fault list, which is frequently scanned, must be explicitly stored in memory. Since only the good machine is evaluated, and the faulty machines are deduced, all the faulty machines implicitly have the same timing characteristics as the good machine, so representation of timing faults is very cumbersome. One final difficulty with deductive simulation is that the deduction equations are data dependent and must be derived for each stimulus. The derivation of these equations is simple for traditional gates, but becomes much more complicated (and time consuming) for more complex modules. Expansion of the fault algebra aggravates all of these problems.

In comparison, the disadvantages of concurrent simulation are that it copies the entire machine state each time a new fault is activated. Copying the entire machine state is expensive in terms of both memory and time. Since the different machines are completely independent, concurrent simulation can represent timing faults as easily as level or logical faults.

3. HIERARCHICAL FAULT SIMULATION SYSTEM

The simulation system developed to achieve the research goals, consists of three major parts: two preprocessors for the fault library and circuit source, and an evaluator which performs the fault simulation. The preprocessors, constructed with LEX [20] , a program for generating lexical analyzers, and YACC [21] , a program for generating parsers, parse their respective source files and produce data structures for the evaluator. The internal data structures are constructed as the source file is read and the result is a compact, linked list data structure (directed acyclic graph), with links along all the paths the simulator is expected to need. The resulting data structure is then transformed into a relocatable structure by making all pointers relative to the base of the data structure. This structure is then written to a file for later use by the evaluator.

The data structures were parsed in separate programs for several reasons. First, under UNIX*, it is difficult to call two YACC-generated parsers from the same program because YACC gives all parsers the same name. One of the parsers could be renamed by editing the YACC output file, but this approach adds another step in the edit-compile-test cycle and is undesirable from a maintainability standpoint. Second, there is no need to reprocess both the circuit and the fault library if only one

*UNIX is a Trademark of Bell Laboratories.

of the two has changed. For small circuits or fault libraries, this extra overhead is negligible, but for larger systems the overhead may be significant. Third, for stylistic and maintenance purposes, it is much easier to cope with three specialized programs than with one large conglomeration. In the following sections each part of the simulator is discussed in detail.

3.1. Fault Library

This section focuses on the fault library segment of the simulator. The intent is to separate the core evaluation routines of the simulator from the details of the fault model. This separation allows the simulator to be used for different technologies without modification, and allows easy expansion of the simulator primitive set. An algorithm for combining primitives to form new primitives has been developed which will be presented in future research.

The fault library represents the precomputation and orderly storage of fault syndromes for all primitive elements. These syndromes may be computed for any fault with a logically modelled effect at the output of the primitive. This means either the wrong logical value, or the correct value at the wrong time. This allows a more comprehensive fault model than the traditional stuck-at fault model which has been proven inadequate [22,23,24]. Precomputation of the fault syndromes decouples the simulator from the fault model in the sense that the fault model is embedded in the primitive library instead of in the simulator evaluation routines. Thus different technologies with different fault models can

be incorporated in the simulation system by building a primitive library of fault syndromes which correspond to that technology. This method is, of course, sensitive to the completeness of the fault library entries for the primitives.

The preprocessor for the fault library parses the source for the fault library and produces a relocatable linked list data structure for the evaluator. The input is structured in a simple LR(0) grammar consisting of about eight keywords, four separators, alphanumeric symbols, and four-valued signal vectors. The case of alphabetic characters is significant, and the parser ignores blanks, tabs, new lines, and " /* comments */". A sample entry for a three input AND gate is shown in Figure 2. Each entry in the fault library begins with the keyword PRIMITIVE, followed by an equal sign, then the name of the primitive. This name is followed by INPUT, OUTPUT, and FAULTLIST sections. The input

```
PRIMITIVE=AND3
INPUT 3 : 1=A,2=B,3=C
OUTPUT 4 : 4=FAND
FAULTLIST 7 :
A1,FAND1          011>
B1,FAND1          101>
C1,FAND1          110>
A0,B0,C0,FAND0    111<
FAND1             0??>
FAND1             ?0?>
FAND1             ??0>
```

Figure 2. Fault Library Source for a 3 Input AND Gate

and output sections have similar structure, the keyword INPUT or OUTPUT followed by a number, then a list of signal names and their corresponding positions in the I/O vector. These assignments have the form "signal number = signal name", and are separated by commas. The number following the keyword is the number of signals in the following list and is used by the parser to allocate storage in advance for the signal list. The keyword FAULTLIST (if present) is followed by the number of vectors following the keyword for the same purpose.

The vectors following the keyword FAULTLIST are composed of tuples, a list of faults covered by the vector and a signal vector composed of logic values. In the current implementation there are five logic values:

```
1 logic one
0 logic zero
> error, should be zero but is one under fault
< error, should be one but is zero under fault
? don't care
% unknown
```

The exact representation of the logic values is immaterial; these symbols were chosen for ease of parsing, and they are intended for internal use only. The list of faults included in the tuple has no meaning to the evaluator. It is completely up to the user to establish any desired fault naming convention. The set of faults which can produce an error on a particular line for the current input is indistinguishable [25,26], so the evaluator does not care whether the fault name list associated with a vector represents one or many faults. The simulator treats each

list of fault names uniquely even if there exist other strings with the same lexical value (sequence of symbols). This uniqueness is required since two faults may have identical names; the uniqueness is established by their location in the circuit hierarchy. In fact, the contents of these name lists are never examined by the simulator.

3.2. Scald Circuit Description

The circuit definition language is a subset of the Structured Computer-Aided Logic Design language (SCALD) [27]. This language was chosen because it is a hierarchical circuit description language and previous work at the University of Illinois implemented SCALD output from the graphics editor DRAW [28]. In SCALD, the circuit is hierarchically defined in terms of macromodules which are defined in terms of other macromodules and/or simulator primitives. Macromodules and primitives may have multiple inputs and outputs. Figure 3 shows the SCALD definition of an Exclusive Or.

```
MNAME=XOR;
PARAMETER=a,b,c;
INV(LOC=XOR1)(A=a,FINV=NUL%00001);
INV(LOC=XOR2)(A=b,FINV=NUL%00002);
NAND2(LOC=XOR3)(A=a,B=NUL%00002,FNAND=NUL%00003);
NAND2(LOC=XOR4)(A=b,B=NUL%00001,FNAND=NUL%00004);
NAND2(LOC=XOR5)(A=NUL%00003,B=NUL%00004,FNAND=c);
END;
```

Figure 3. SCALD Definition of an Exclusive Or Module

The module name follows the keyword MNAME and is terminated with a semicolon. The next line beginning with the keyword PARAMETER enumerates the input-output lines; order is unimportant. Most simulators are based on gate level primitives with only one output, this is not adequate since many technologies permit structures which have no gate equivalent. The multiple input/multiple output capability of SCALD is much more powerful in this respect. The parameters (signals) are followed by calls to other modules, either macromodules or primitives. These calls consist of the module name followed by a unique location and then the list of signal bindings. The location distinguishes between several calls to the same module and the signal bindings provide correspondences between the signal names in the called module and the signal names in the calling module. The module calls are followed by the keyword END to signify the end of the current module definition.

This hierarchical description is compact because each module is defined only once but can be called many times. This representation is much more compact than expanding the circuit to the lowest level (simulator primitives). This compactness provides better locality, which is important for good cache miss and page fault ratios, since the circuit description is constantly scanned by the evaluator.

3.3. The Evaluator

The evaluator is the core of the fault simulation system. This program reads the data structures produced by the two preprocessors and relocates each according to its base address. Then some additional

linking is performed to link SCALD calls to primitives to the appropriate fault library definition of those primitives, and to link the primitives to a functional evaluation routines. Once this process is complete, the evaluator is ready to apply vectors to the circuit.

The evaluation begins by applying the input vector to the highest level scald module, which must encompass the entire circuit. The evaluator reorders the vector and proceeds to call itself recursively through macromodule calls until a simulator primitive is encountered. The result of this evaluation is then applied to the next higher module and another call at that level is given to the evaluator. This process continues until all the pending activity at the current level is completed, then the evaluator returns the result to the next higher level. This process implements a depth-first evaluation of the circuit.

The evaluation process consists of two major portions, macromodule evaluation and primitive evaluation. The macromodule evaluation occurs first and is the simplest so it will be discussed first.

3.4. Macromodule Evaluation

Macromodule evaluation consists of two parts, choosing the next available module call to evaluate and reordering the signal vector for that call. If a scheduling algorithm is used, the order of evaluation may not correspond to the static ordering in the module definition, since some signals may be undefined (internal nodes), and some modules may be evaluated more than once in sequential circuits. When all the modules have been evaluated and there is no more internal signal

activity, the macromodule evaluation process terminates by returning to the calling level, with updated external signal values. These values may be scheduled for application to the circuit at some time in the future.

3.5. Primitive Evaluation

Primitive evaluation is somewhat more complicated, but also a two step process, fault activation in the current module and error propagation from the inputs to the outputs of the current module.

3.5.1. Fault Injection

The fault injection process is characterized by table lookup for matches in the fault library with the current input vector. These matches are calculated with a matching function that resolves disparate values for the same signal into matches or differences. This function is responsible for matching with don't cares and unknowns. If one or more vectors are matched from the fault library, the corresponding fault lists are attached, along with the complete path to the current module, to all outputs which evidence the errors. A fault (list) may appear on more than one output, which complicates the task of fault propagation.

3.5.2. Fault Propagation

Once fault activation is complete, the input signals are scanned for attached fault lists. If the signals contain fault lists, these lists are decomposed into sets characterized by unique error syndromes at the inputs of the current module. The decomposition is done with a

double hashing process: first, the error is hashed according to its name and location, then according to the input(s) on which it occurs. The result of this hashing is an inverted list where all of the fault lists with the same error syndrome are collapsed into a single fault list. Each of the unique error syndromes is then evaluated with the functional model of the primitive and the results compared to the good module outputs. If the outputs differ, then the errors creating the current syndrome propagate on all outputs which differ from the good machine. Propagation is effected by attaching the list of fault lists to the appropriate outputs. If the syndrome creates an output which is identical to the good machine, then the errors are marked as potentially absorbed (they may be propagated elsewhere) for later processing. Each syndrome is evaluated in turn until all syndromes are exhausted. In the worst case, the number of syndromes is equal to the input range of the module, but in practice it should be only a small fraction. The most significant feature of this process is that the maximum possible collapsing is performed on error syndromes because they are dynamically collapsed.

4. FAULT SIMULATION EXAMPLES

In this chapter the fault simulation algorithm is further clarified through examples. The first example, an Exclusive Or circuit, demonstrates the details of the algorithm. The second example, a fast multiplier, shows how the simulator behaves with realistic circuits and indicates how well the simulator performs.

4.1.1. Exclusive Or

In this section an example of the simulation algorithm is presented using an Exclusive Or circuit. This circuit, shown in Figure 4, is a multiple input, single output module, composed of gate-level primitives. Although using a gate-level description does not fully utilize the capabilities of the simulator, it is easy to follow.

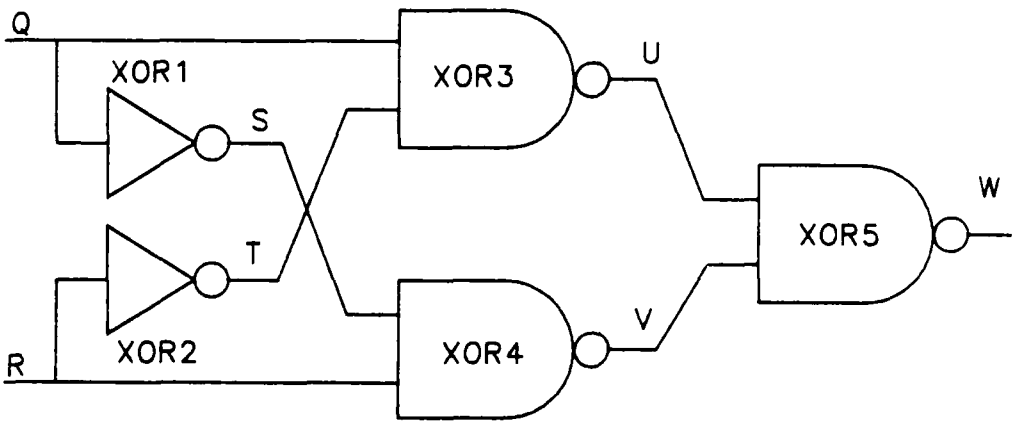


Figure 4. Exclusive Or Circuit for Simulation Example

The notation used in this example is simple, the gates are referred to by location, i.e., XOR1, and the circuit nodes are labelled, i.e., T. Fault names are not required to be unique, and as seen in this example, a unique location is required to fully qualify fault names. Lists of indistinguishable faults are enclosed in parentheses with the unique location beginning the list, and lists of these fault lists are enclosed in square brackets to denote a fault list. The simulator does not store the lists or manipulate the data in exactly this form; the notation is intended for clarity. Some details of the simulation have been left out, but this example illustrates the essence of the simulation algorithm.

Let the input vector take the value 10 on inputs QR. The scheduler determines that only modules (in this case primitives) XOR1 and XOR2 have completely known inputs; all other modules have some unknown inputs. The scheduler chooses to evaluate module XOR1 first because it is the first module in the list of evaluable modules. Evaluating XOR1 for the input Q=1 determines that the output S=0; this constitutes the good machine evaluation. Next the fault library entry for this type of module (inverter) is searched for matches with the input vector. This is the activation phase of fault simulation. One match is found and this match is attached to the output vector for module XOR1. This attachment is denoted by the signal name and value followed by the list of fault names:

S=0 [(XOR1,A0,FINV1)]

There are no faults attached to the input so the fault propagation phase

is skipped. The evaluation is completed by scheduling the change in S to occur at the appropriate time in the future. There is only one module XOR2 left on the evaluation list, so XOR2 is evaluated next. The input R=0 produces the output T=1 and the fault library search yields one match,

T=1 [(XOR2,A1,FINV0)]

Again there are no faults attached to the input so the fault propagation phase is skipped, and the output T is scheduled to change at the appropriate time in the future. In this example the changes in S and T are assumed to occur at the same time.

Since the evaluation list is empty, the simulator clock advances to the next signal event, where the changes in S and T are applied. The scheduler checks all modules affected by these changes and finds that modules XOR3 and XOR4 may now be evaluated. The good machine evaluation of XOR3 determines that for the input vector QT=11, the output U=0. Searching the fault library entry for a 2 input NAND produces one match,

U=0 [(XOR3,A0,B0,FNAND1)]

The input Q does not have any attached faults, but the input T does, so these faults must be checked for propagation through the module. Since there is only one input with attached faults, no fault collapsing is possible so this step is skipped. Next, the (only) error syndrome is synthesized QT'=10 and applied to the module, which determines that U'=1. Since U' and U differ propagation occurs, and the fault list associated with this fault syndrome is attached to the output U. The

resulting fault list attached to U is then

U=0 [(XOR3,A0,B0,FNAND1)(XOR2,A1,FINV0)]

As before the evaluation ends by scheduling the output change at the appropriate time in the future.

The evaluation of XOR4 occurs in a similar fashion. The input vector RS=00 produces an output of V=1, and there is one match in the fault library,

V=1 [(XOR4,FNAND0)]

Again, only one input contains a fault list, so no collapsing is done, and the one error syndrome, RS'=01, produces an output V'=1. Since V = V', the fault list associated with the error syndrome is not propagated, but marked as "potentially" absorbed. Fanout elsewhere in the circuit could have allowed other paths for the faults to propagate, so the final determination is delayed until all circuit activity has ceased. In this case it is easy to see that there is no fanout for this list so the fault list is completely absorbed.

Again, the evaluation queue is empty so the simulation clock is advanced to the next signal event where the values for U and V are changed. These signals affect XOR5 so this module is scheduled for evaluation. The good machine evaluation of XOR5 for the input vector UV=01 produces the output W=1, and the activation phase finds one match in the fault library,

W=1 [(XOR5,A1,FNAND0)]

Both inputs U and V have attached fault lists so fault collapsing is

applied to the input vector. The fault lists are disjoint so no collapsing occurs. The two lists of faults produce two error syndromes,

UV'=11 from [(XOR3,A0,B0,FNAND1)] attached to U
UV"=00 from [(XOR4,FNAND0)] attached to V

Since the simulator operates under the single fault assumption, there are no other syndromes possible. Evaluation of the first syndrome UV'=11 produces an output of W'=0 which differs from W, so the associated fault list is attached to W. The second syndrome UV"=00 produces an output of W"=1 which is the same as W so the associated fault list does not propagate, and this list is flagged as potentially absorbed. The final fault list attached to W is then

W=1 [(XOR5,A1,FNAND0)(XOR3,A0,B0,FNAND1)(XOR2,A1,FINV0)]

The evaluation of XOR5 ends by scheduling W to change at some future time. The queue is again empty so the simulation clock advances to the only remaining event, and the new value for W is applied. This node has no fanout to any other modules so no modules are scheduled for evaluation. The event queue and the evaluation queue are now empty so simulation activity is ready to terminate.

The final task remaining is to check the fault list attached to the primary output against the list of potentially absorbed faults to see which faults were propagated by alternate paths. Faults which appear in both lists are removed from the list of potentially absorbed faults since they are proven to be observable, and the remaining list is reported as absorbed faults. In this example the list of absorbed faults is

[(XOR1,A0,FINV1)] at XOR4
[(XOR4,FNAND0)] at XOR5

This information about absorbed faults is useful to a circuit or test designer since it indicates that the primary input was sufficient to activate these faults, but that they are not observable because propagation was blocked at the listed locations.

4.1.2. Fast Multiplier Example

The following example is a 24 bit by 24 bit fast multiplier* which produces a 48 bit result. This size is appropriate for mantissa multiplication of 32 bit floating point numbers. This design trades space for speed and is quite large; approximately 3 mm by 3 mm, and represents about 30,000 active devices. For simplicity there are no propagate, generate carry signals or carry lookahead. This does not change the functionality, but it does change the speed of the multiplier. The salient features of the design which produce its speed are that a number of independent partial products are generated in parallel and then summed in parallel via several stages of highly vertical adders, with very few carries between adders.

The two 24 bit inputs are divided into 4 bit nibbles, and each combination of nibbles is used to generate one of the 36 partial products. These partial products are summed in three stages of adders; the first two stages are highly vertical, while the last stage is more horizontal.

---

*Portions of this example were provided by the General Electric Corporate Research Center.

The vertical adders are characterized as such because they sum a column of five, 2 bit wide numbers. The horizontal adders sum two, 4 bit wide numbers. One of the horizontal types of adders also accepts a carry input. The carries which normally limit the speed of large additions must be eventually resolved. This resolution occurs in the last stage and requires only eight carries between adder modules. The cellular organization of the multiplier is shown in Figure 5.

The SCALD description of this design represents five levels of hierarchy with 21,000 interconnections and 4500 instances of primitives. The partial SCALD description for this circuit is given in Appendix A. The first level consists of calls to the four types of macromodules pre-

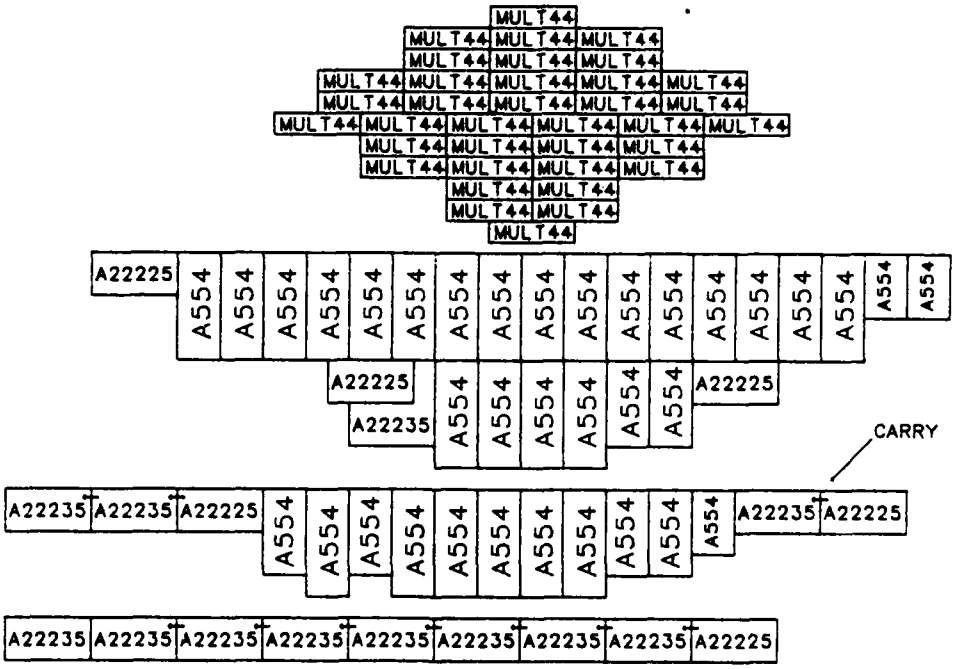


Figure 5. Cellular Organization of the Fast Multiplier

viously described. These modules consist of calls to other macromodules, which eventually lead to calls to the simulator primitives.

The 4 by 4 bit multiplier used to generate the partial products is a combinational multiplier similar to that shown in Hayes [29]. The multiplier consists of two major parts; an AND array which generates 2 bit partial products and an adder array to sum these products. See Figures 6 and 7 for more detail.

The partial products generated by these multipliers are summed in three stages. Each stage is a mixture of three types of adders. The first type is a vertical adder which adds five, 2 bit wide numbers. The easiest way to think of the operation of this adder that it produces the binary weighted sum of the two columns, or the sum of the right column

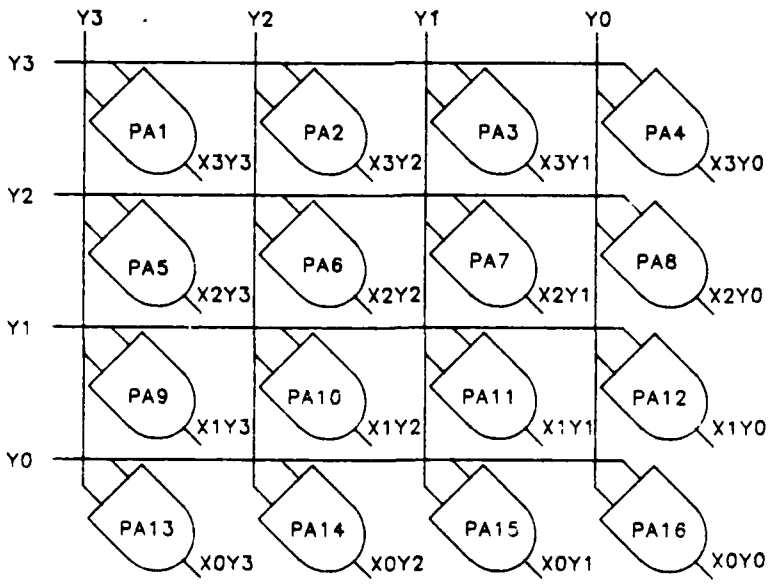


Figure 6. AND Array for Fast Multiplier Example

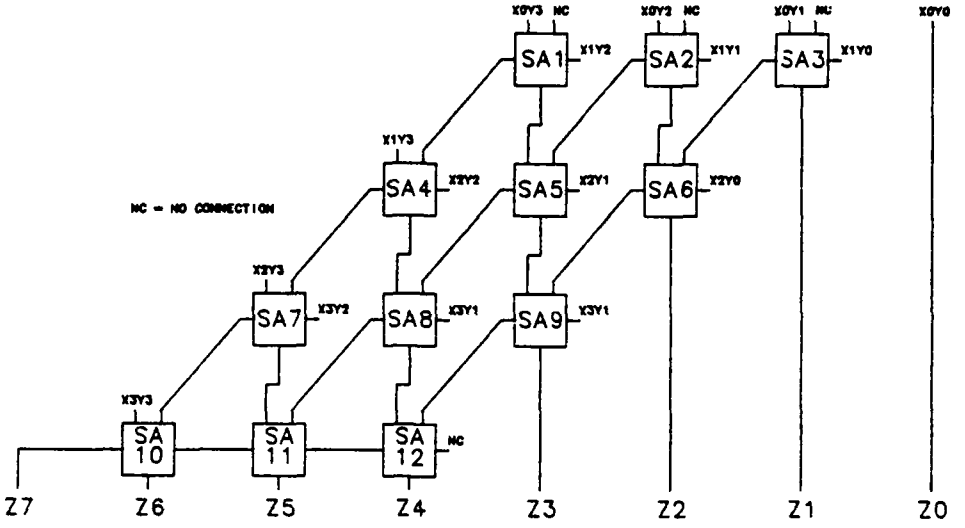


Figure 7. Sum Array for Fast Multiplier Example

of bits summed with the weighted sum (times 2) of the left column. While not the largest module, the design for this module is the most confusing. This design is shown in Figure 8. The final two modules consist of chains of full and half adders. Their operation is obvious as shown in Figures 9 and 10.

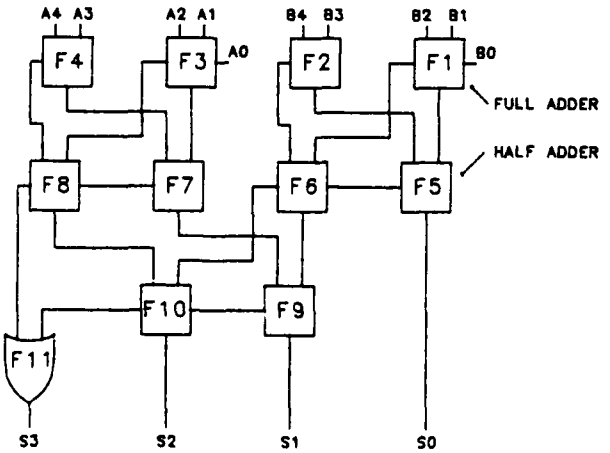


Figure 8. 554 Adder for Fast Multiplier Example

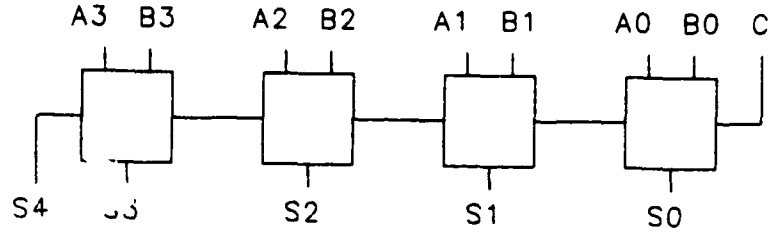


Figure 9. 22235 Adder for Fast Multiplier Example

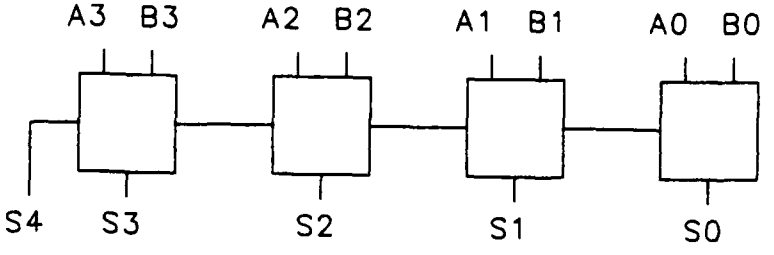


Figure 10. 22225 Adder for Fast Multiplier Example

Using the system profiler [30] the simulator was found to spend most of its time in fault propagation and garbage collection. Although performance of the simulator depends on circuit topology, input vector, and number of faults propagated, approximate performance can be averaged over a wide range of circuits, for many input vectors. For the multiplier the simulator processed the 4500 primitive calls (one pass) in an average of 45 seconds per input vector (60 seconds with profiling). The

size of the simulation in memory which depends on these same attributes stablized at about 1.5M bytes. The memory allocation breaks down as 88K bytes for the simulator code, 5K bytes for the fault library, 54K bytes for the SCALD description, and the rest is workspace for fault activity and circuit evaluation.

5. DISCUSSION AND CONCLUSIONS

The statistics presented in Chapter 4 are very encouraging for several reasons. First, the system description can be represented in a very compact form, and non-faulting simulation measurements show the overhead for walking the hierarchy is insignificant compared to fault simulation. Second, the fault library is small and the vector lookup from the library is quite fast. The library can be kept small by carefully choosing its contents. The execution times represent 1-2 orders of magnitude speed improvement over the TEGAS fault simulator for a similar circuit. Finally, the workspace is large but quite acceptable for a large circuit, like the Fast Multiplier example, especially since no minimization techniques such as data packing have been applied.

Experience with the simulation system has shown several major advantages to this approach. The compactness of the hierarchical circuit description is important during execution because it significantly reduces the run-time memory requirements. The complete design, entry, and debug cycle for the fast multiplier took only two days. The design was entered in a top-down fashion. Functional descriptions were provided for each type of module and the highest level description was debugged. Then each of the modules was defined in more detail, and these descriptions were debugged. This define-debug cycle continued until complete hierarchy was entered and debugged. The author found

that the hierarchical data structures inside the simulator aided the development of user-friendly reporting for debugging. In debugging mode the simulator reports the position in the hierarchy by dumping the module call stack. The simulator also reports the I/O vector to the current module by printing one of the string parameters to the current evaluation. This contrasts with the complicated number-to-name and table lookup often required to do similar reporting for flat system descriptions.

The simulation system as presented in this thesis has been implemented. This implementation was sufficient to design and test the basic algorithm and with enhancements promises to become a complete and usable fault simulator. During the development of this simulator the author found that there were a few disadvantages to simulating from a purely hierarchical data structure. Specifically, there is some information which is unique to each instance of a module and cannot be stored in the hierarchy. For instance, state information is necessary for each instance of a sequential module. Since the hierarchy is unsuitable for storing state, some other alternative structure must be used. The author chose a tree data structure for this unique information because it is similar to the hierarchy, and there is a convenient mapping from one structure to the other. In this scheme the simulator can walk the hierarchy to access invariant information, and walk the tree structure to access unique information for each module.

While state information is the most obvious use of the tree, it is well suited to logging detected faults and removing them from further consideration. Along with the state vector for each module, the tree also contains a fault vector with one entry for each vector in the fault library for that type of module. This vector can be used to mark faults when they are detected at the primary outputs, and remove them from further consideration. This marking substantially improves simulator performance by reducing the number of faults under consideration in future vectors, which increases the effective execution speed.

This marking can also be used by the user to control which faults are considered by the simulation. If the user is interested in the fault coverage of a particular module or type of module but not the rest of the circuit, then by marking the faults in these modules as undetected and all others as detected, the simulator will inject only these faults. The user can then develop a test set for the circuit module by module, and prevent the simulator from considering faults in modules that have already been analyzed. This technique is a manual form of partitioning that allows the user to maintain the full capability of the simulator while achieving much better speed by controlling the fault injection.

6. FUTURE WORK

The fault simulation techniques presented in this thesis are far from complete. The preliminary performance results indicate that these techniques warrant further exploration. The next step should be the implementation of tree data structures and fault removal (as faults are detected) to increase simulator speed. Then signal state retention should be added to the tree in preparation for sequential capability. Evaluation of sequential systems and nominal delays should be added by implementing an event queue and activity-directed evaluation. The implementation of these capabilities will complete the development of the fault simulator and should be followed by a thorough performance analysis.

Once this stage of development is complete, the emphasis should shift to using this fault simulator as a host for test generation research. Current research suggests that the additional information about a system which is available in the hierarchy can be used with heuristic algorithms and expert systems to automate test generation. As integrated circuits get more complicated, brute force techniques for test generation become less feasible. Greater emphasis must be placed on test generation systems that use intelligence and sophistication to reduce the computational overhead and produce higher quality tests.

APPENDIX A. SCALD SOURCE FOR THE FAST MULTIPLIER

This is the SCALD source for the fast multiplier. The listing begins with the definiton of the highest level and proceeds by defining each module in greater detail until the simulator primitives are called at the lowest level.

```
MNAME=FASTMULT;
PARAMETER=L3,L2,L1,L0,K3,K2,K1,K0,J3,J2,J1,J0,
          I3,I2,I1,I0,H3,H2,H1,H0,G3,G2,G1,G0,
          F3,F2,F1,F0,E3,E2,E1,E0,D3,D2,D1,D0,
          C3,C2,C1,C0,B3,B2,B1,B0,A3,A2,A1,A0,
          M47,M46,M45,M44,M43,M42,M41,M40,M39,M38,M37,M36,
          M35,M34,M33,M32,M31,M30,M29,M28,M27,M26,M25,M24,
          M23,M22,M21,M20,M19,M18,M17,M16,M15,M14,M13,M12,
          M11,M10,M9,M8,M7,M6,M5,M4,M3,M2,M1,M0,GND;
MULT44(LOC=F1)(A3=L3,A2=L2,A1=L1,A0=L0,B3=F3,B2=F2,B1=F1,B0=F0,
       M7=NUL%000,M6=NUL%001,M5=NUL%002,M4=NUL%003,M3=NUL%004,
       M2=NUL%005,M1=NUL%006,M0=NUL%007,GND=GND);
MULT44(LOC=F2)(A3=L3,A2=L2,A1=L1,A0=L0,B3=E3,B2=E2,B1=E1,B0=E0,
       M7=NUL%008,M6=NUL%009,M5=NUL%010,M4=NUL%011,M3=NUL%012,
       M2=NUL%013,M1=NUL%014,M0=NUL%015,GND=GND);
MULT44(LOC=F3)(A3=K3,A2=K2,A1=K1,A0=K0,B3=F3,B2=F2,B1=F1,B0=F0,
       M7=NUL%016,M6=NUL%017,M5=NUL%018,M4=NUL%019,M3=NUL%020,
       M2=NUL%021,M1=NUL%022,M0=NUL%023,GND=GND);
MULT44(LOC=F4)(A3=L3,A2=L2,A1=L1,A0=L0,B3=D3,B2=D2,B1=D1,B0=D0,
       M7=NUL%024,M6=NUL%025,M5=NUL%026,M4=NUL%027,M3=NUL%028,
       M2=NUL%029,M1=NUL%030,M0=NUL%031,GND=GND);
                          .
                          .
                          .
A554(LOC=F37) /* 0 */
    (A4=NUL%120,A3=NUL%128,A2=NUL%136,A1=NUL%144,A0=NUL%152,
     B4=NUL%121,B3=NUL%129,B2=NUL%137,B1=NUL%145,B0=NUL%153,
     S3=NUL%284,S2=NUL%285,S1=NUL%286,S0=NUL%287);
A554(LOC=F38) /* 1 */
    (A4=NUL%122,A3=NUL%130,A2=NUL%138,A1=NUL%146,A0=NUL%154,
     B4=NUL%123,B3=NUL%131,B2=NUL%139,B1=NUL%147,B0=NUL%155,
     S3=NUL%288,S2=NUL%289,S1=NUL%290,S0=NUL%291);
A554(LOC=F39) /* 2 */
    (A4=NUL%124,A3=NUL%132,A2=NUL%140,A1=NUL%148,A0=NUL%156,
```

```
     B4=NUL%125,B3=NUL%133,B2=NUL%141,B1=NUL%149,B0=NUL%157,
     S3=NUL%292,S2=NUL%293,S1=NUL%294,S0=NUL%295);
A554(LOC=F40) /* 3 */
    (A4=NUL%126,A3=NUL%134,A2=NUL%142,A1=NUL%150,A0=NUL%158,
     B4=NUL%127,B3=NUL%135,B2=NUL%143,B1=NUL%151,B0=NUL%159,
     S3=NUL%296,S2=NUL%297,S1=NUL%298,S0=NUL%299);
                          .
                          .
                          .
A22225(LOC=F61)(A3=NUL%008,A2=NUL%009,A1=NUL%010,A0=NUL%011,
/* x */ B3=NUL%016,B2=NUL%017,B1=NUL%018,B0=NUL%019,
        S4=NUL%378,S3=NUL%379,S2=NUL%380,S1=NUL%381,S0=NUL%382);
A22225(LOC=F62)(A3=NUL%039,A2=NUL%088,A1=NUL%089,A0=NUL%090,
/* y */ B3=NUL%047,B2=NUL%096,B1=NUL%097,B0=NUL%098,
        S4=NUL%383,S3=NUL%384,S2=NUL%385,S1=NUL%386,S0=NUL%387);
A22225(LOC=F63)(A3=NUL%264,A2=NUL%265,A1=NUL%266,A0=NUL%267,
/* z */ B3=NUL%272,B2=NUL%273,B1=NUL%274,B0=NUL%275,
        S4=NUL%388,S3=NUL%389,S2=NUL%390,S1=NUL%391,S0=NUL%392);
A22235(LOC=F64)(A3=NUL%104,A2=NUL%105,A1=NUL%106,A0=NUL%107,
/* * */ B3=NUL%112,B2=NUL%113,B1=NUL%114,B0=NUL%115,C0=NUL%099,
       S4=NUL%393,S3=NUL%394,S2=NUL%395,S1=NUL%396,S0=NUL%397);
/*  next level  */
A22225(LOC=F80)(A3=NUL%348,A2=NUL%349,A1=NUL%350,A0=NUL%351,
/* f */ B3=NUL%346,B2=NUL%347,B1=NUL%352,B0=NUL%353,
        S4=NUL%460,S3=M9,S2=M8,S1=M7,S0=M6);
A22235(LOC=F79)(A3=NUL%340,A2=NUL%341,A1=NUL%342,A0=NUL%343,
/* e */ B3=NUL%338,B2=NUL%339,B1=NUL%344,B0=NUL%345,C0=NUL%460,
        S4=NUL%457,S3=NUL%458,S2=NUL%459,S1=M11,S0=M10);
A554(LOC=F78) /* d */
    (A4=NUL%334,A3=NUL%336,A2=GND,A1=NUL%389,A0=GND,
     B4=NUL%335,B3=NUL%337,B2=GND,B1=NUL%390,B0=GND,
     S3=NUL%453,S2=NUL%454,S1=NUL%455,S0=NUL%456);
A554(LOC=F77) /* c */
    (A4=NUL%332,A3=NUL%322,A2=NUL%330,A1=GND,A0=GND,
     B4=NUL%333,B3=NUL%323,B2=NUL%331,B1=NUL%388,B0=GND,
     S3=NUL%449,S2=NUL%450,S1=NUL%451,S0=NUL%452);
                          .
                          .
                          .
A22235(LOC=F66)(A3=NUL%004,A2=NUL%005,A1=NUL%354,A0=NUL%355,
/* 1 */ B3=NUL%379,B2=NUL%380,B1=NUL%381,B0=NUL%382,C0=NUL%408,
        S4=NUL%403,S3=NUL%404,S2=NUL%405,S1=NUL%406,S0=NUL%407);
A22235(LOC=F65)(A3=NUL%000,A2=NUL%001,A1=NUL%002,A0=NUL%003,
/* 0 */ B3=GND,B2=GND,B1=GND,B0=NUL%378,C0=NUL%403,
        S4=NUL%398,S3=NUL%399,S2=NUL%400,S1=NUL%401,S0=NUL%402);
/* final stage of adders */
A22225(LOC=F89)(A3=GND,A2=NUL%457,A1=NUL%458,A0=NUL%459,
/* 8 */ B3=NUL%455,B2=NUL%456,B1=NUL%391,B0=NUL%392,
        S4=NUL%469,S3=M15,S2=M14,S1=M13,S0=M12);
```

```
A22235(LOC=F88)(A3=NUL%449,A2=NUL%450,A1=NUL%451,A0=NUL%452,
/* 7 */ B3=NUL%447,B2=NUL%448,B1=NUL%453,B0=NUL%454,C0=NUL%469,
       S4=NUL%468,S3=M19,S2=M18,S1=M17,S0=M16);
                          .
                          .
                          .
A22225(LOC=F81)(A3=NUL%399,A2=NUL%400,A1=NUL%401,A0=NUL%402,
/* 0 */ B3=GND,B2=GND,B1=GND,B0=NUL%462,
       S4=NUL%461,S3=M47,S2=M46,S1=M45,S0=M44);
END;

MNAME=A22235;
PARAMETER=A3,A2,A1,A0,B3,B2,B1,B0,C0,S4,S3,S2,S1,S0;
FA(LOC=A22235_1)(A=A0,B=B0,CIN=C0,COUT=NUL%000,SUM=S0);
FA(LOC=A22235_2)(A=A1,B=B1,CIN=NUL%000,COUT=NUL%001,SUM=S1);
FA(LOC=A22235_3)(A=A2,B=B2,CIN=NUL%001,COUT=NUL%002,SUM=S2);
FA(LOC=A22235_4)(A=A3,B=B3,CIN=NUL%002,COUT=S4,SUM=S3);
END;

MNAME=A22225;
PARAMETER=A3,A2,A1,A0,B3,B2,B1,B0,S4,S3,S2,S1,S0;
HA(LOC=A22225_1)(A=A0,B=B0,COUT=NUL%000,SUM=S0);
FA(LOC=A22225_2)(A=A1,B=B1,CIN=NUL%000,COUT=NUL%001,SUM=S1);
FA(LOC=A22225_3)(A=A2,B=B2,CIN=NUL%001,COUT=NUL%002,SUM=S2);
FA(LOC=A22225_4)(A=A3,B=B3,CIN=NUL%002,COUT=S4,SUM=S3);
END;

MNAME=HA;
PARAMETER=A,B,COUT,SUM;
XOR2(LOC=HA1)(A=A,B=B,FXOR=SUM);
AND2(LOC=HA2)(A=A,B=B,FAND=COUT);
END;

MNAME=FA;
PARAMETER=A,B,CIN,COUT,SUM;
XOR3(LOC=FA1)(A=A,B=B,C=CIN,FXOR=SUM);
AND2(LOC=FA2)(A=B,B=CIN,FAND=NUL%000);
AND2(LOC=FA3)(A=A,B=CIN,FAND=NUL%001);
AND2(LOC=FA4)(A=A,B=B,FAND=NUL%002);
OR3(LOC=FA5)(A=NUL%002,B=NUL%001,C=NUL%000,FOR=COUT);
END;

MNAME=A554;
PARAMETER=A4,A3,A2,A1,A0,B4,B3,B2,B1,B0,S3,S2,S1,S0;
FA(LOC=A554_1)(A=B2,B=B1,CIN=B0,
       COUT=NUL%001,SUM=NUL%000);
HA(LOC=A554_2)(A=B4,B=B3,
       COUT=NUL%003,SUM=NUL%002);
FA(LOC=A554_3)(A=A2,B=A1,CIN=A0,
```

```
        COUT=NUL%005,SUM=NUL%004);
HA(LOC=A554_4)(A=A4,B=A3,
        COUT=NUL%007,SUM=NUL%006);
HA(LOC=A554_5)(A=NUL%002,B=NUL%000,
        COUT=NUL%008,SUM=S0);
FA(LOC=A554_6)(A=NUL%003,B=NUL%001,CIN=NUL%008,
        COUT=NUL%010,SUM=NUL%009);
HA(LOC=A554_7)(A=NUL%006,B=NUL%004,
        COUT=NUL%012,SUM=NUL%011);
FA(LOC=A554_8)(A=NUL%007,B=NUL%005,CIN=NUL%012,
        COUT=NUL%014,SUM=NUL%013);
HA(LOC=A554_9)(A=NUL%011,B=NUL%009,
        COUT=NUL%015,SUM=S1);
FA(LOC=A554_10)(A=NUL%013,B=NUL%010,CIN=NUL%015,
        COUT=NUL%016,SUM=S2);
OR2(LOC=A554_11)(A=NUL%014,B=NUL%016,C=S3);
END;

MNAME=PARRAY4X4;
PARAMETER=X3,X2,X1,X0,Y3,Y2,Y1,Y0,
        X3Y3,X3Y2,X3Y1,X3Y0,X2Y3,X2Y2,X2Y1,X2Y0,
        X1Y3,X1Y2,X1Y1,X1Y0,X0Y3,X0Y2,X0Y1,X0Y0;
AND2(LOC=PA1)(A=X3,B=Y3,FAND=X3Y3);
AND2(LOC=PA2)(A=X3,B=Y2,FAND=X3Y2);
AND2(LOC=PA3)(A=X3,B=Y1,FAND=X3Y1);
AND2(LOC=PA4)(A=X3,B=Y0,FAND=X3Y0);
AND2(LOC=PA5)(A=X2,B=Y3,FAND=X2Y3);
AND2(LOC=PA6)(A=X2,B=Y2,FAND=X2Y2);
AND2(LOC=PA7)(A=X2,B=Y1,FAND=X2Y1);
AND2(LOC=PA8)(A=X2,B=Y0,FAND=X2Y0);
AND2(LOC=PA9)(A=X1,B=Y3,FAND=X1Y3);
AND2(LOC=PA10)(A=X1,B=Y2,FAND=X1Y2);
AND2(LOC=PA11)(A=X1,B=Y1,FAND=X1Y1);
AND2(LOC=PA12)(A=X1,B=Y0,FAND=X1Y0);
AND2(LOC=PA13)(A=X0,B=Y3,FAND=X0Y3);
AND2(LOC=PA14)(A=X0,B=Y2,FAND=X0Y2);
AND2(LOC=PA15)(A=X0,B=Y1,FAND=X0Y1);
AND2(LOC=PA16)(A=X0,B=Y0,FAND=X0Y0);
END;

MNAME=SARRAY4X4;
PARAMETER=X3Y3,X3Y2,X3Y1,X3Y0,X2Y3,X2Y2,X2Y1,X2Y0,
        X1Y3,X1Y2,X1Y1,X1Y0,X0Y3,X0Y2,X0Y1,X0Y0,
        Z7,Z6,Z5,Z4,Z3,Z2,Z1,Z0,GND;
WIRE(LOC=SA1)(A=X0Y0,B=Z0);
FA(LOC=SA2)(A=X0Y1,B=GND,CIN=X1Y0,COUT=NUL%004,SUM=Z1);
FA(LOC=SA3)(A=X0Y2,B=GND,CIN=X1Y1,COUT=NUL%002,SUM=NUL%003);
FA(LOC=SA4)(A=X0Y3,B=GND,CIN=X1Y2,COUT=NUL%000,SUM=NUL%001);
FA(LOC=SA5)(A=NUL%003,B=NUL%004,CIN=X2Y0,COUT=NUL%009,SUM=Z2);
```

```
FA(LOC=SA6)(A=NUL%001,B=NUL%002,CIN=X2Y1,COUT=NUL%007,SUM=NUL%008);
FA(LOC=SA7)(A=X1Y3,B=NUL%000,CIN=X2Y2,COUT=NUL%005,SUM=NUL%006);
FA(LOC=SA8)(A=NUL%008,B=NUL%009,CIN=X3Y0,COUT=NUL%014,SUM=Z3);
FA(LOC=SA9)(A=NUL%006,B=NUL%007,CIN=X3Y1,COUT=NUL%012,SUM=NUL%013);
FA(LOC=SA10)(A=X2Y3,B=NUL%005,CIN=X3Y2,COUT=NUL%010,SUM=NUL%011);
FA(LOC=SA11)(A=NUL%013,B=NUL%014,CIN=GND,COUT=NUL%016,SUM=Z4);
FA(LOC=SA12)(A=NUL%011,B=NUL%012,CIN=NUL%016,COUT=NUL%015,SUM=Z5);
FA(LOC=SA13)(A=X3Y3,B=NUL%010,CIN=NUL%015,COUT=Z7,SUM=Z6);
END;

MNAME=MULT44;
PARAMETER=A3,A2,A1,A0,B3,B2,B1,B0,M7,M6,M5,M4,M3,M2,M1,M0,GND;
PARRAY4X4(LOC=M1)(X3=A3,X2=A2,X1=A1,X0=A0,Y3=B3,Y2=B2,Y1=B1,Y0=B0,
        X3Y3=NUL%000,X3Y2=NUL%001,X3Y1=NUL%002,X3Y0=NUL%003,
        X2Y3=NUL%004,X2Y2=NUL%005,X2Y1=NUL%006,X2Y0=NUL%007,
        X1Y3=NUL%008,X1Y2=NUL%009,X1Y1=NUL%010,X1Y0=NUL%011,
        X0Y3=NUL%012,X0Y2=NUL%013,X0Y1=NUL%014,X0Y0=NUL%015);
SARRAY4X4(LOC=M2)(
        X3Y3=NUL%000,X3Y2=NUL%001,X3Y1=NUL%002,X3Y0=NUL%003,
        X2Y3=NUL%004,X2Y2=NUL%005,X2Y1=NUL%006,X2Y0=NUL%007,
        X1Y3=NUL%008,X1Y2=NUL%009,X1Y1=NUL%010,X1Y0=NUL%011,
        X0Y3=NUL%012,X0Y2=NUL%013,X0Y1=NUL%014,X0Y0=NUL%015,
        Z7=M7,Z6=M6,Z5=M5,Z4=M4,Z3=M3,Z2=M2,Z1=M1,Z0=M0,GND=GND);
END;
```

APPENDIX B. FAULT LIBRARY SOURCE

This listing is the source for the fault library. All the primitives used by the simulator must appear in this library. The fault vector entries are optional and can be removed to turn off fault simulation.

```
PRIMITIVE=WIRE
INPUT 1 : 1=A
OUTPUT 1 : 2=B
DELAY 0

PRIMITIVE=BUF
INPUT 1 : 1=A
OUTPUT 1 : 2=FBUF
DELAY 1
FAULTLIST 2 :
A0,FBUF0        1<
A1,FBUF1        0>

PRIMITIVE=INVERT
INPUT 1 : 1=A
OUTPUT 1 :2=ABAR
DELAY 1
FAULTLIST 2 :
A0,B1           1>
A1,B0           0<

PRIMITIVE=XOR2
INPUT 2 : 1=A,2=B
OUTPUT 1 : 3=FXOR
DELAY 1
FAULTLIST 4 :
A0,FXOR0        10<
B0,FXOR0        01<
A0,B0,FXOR1     11>
A1,B1,FXOR1     00>

PRIMITIVE=XOR3
INPUT 3 : 1=A,2=B,3=C
OUTPUT 1 : 4=FXOR
```

```
DELAY 1
FAULTLIST 8 :
A0,FXOR0          100<
B0,FXOR0          010<
C0,FXOR0          001<
B0,C0,FXOR1       011>
A0,C0,FXOR1       101>
A0,B0,FXOR1       110>
A0,B0,C0,FXOR0    111<
A1,B1,C1,FXOR1    000>

PRIMITIVE=AND2
INPUT 2 : 1=A,2=B
OUTPUT 1 : 3=FAND
DELAY 1
FAULTLIST 4 :
A0,B0,FAND0       11<
A1,FAND1          01>
B1,FAND1          10>
FAND1             00>

PRIMITIVE=AND3
INPUT 3 : 1=A,2=B,3=C
OUTPUT 1 : 4=FAND
DELAY 1
FAULTLIST 7 :
A1,FAND1          011>
B1,FAND1          101>
C1,FAND1          110>
A0,B0,C0,FAND0    111<
FAND1-1           0??>
FAND1-2           ?0?>
FAND1-3           ??0>

PRIMITIVE=AND4
INPUT 4 : 1=A,2=B,3=C,4=D
OUTPUT 1 : 5=FAND
DELAY 1
FAULTLIST  9 :
A1,FAND1                  0111>
B1,FAND1                  1011>
C1,FAND1                  1101>
D1,FAND1                  1110<
A0,B0,C0,D0,FAND0         1111<
FAND1-1                   0???>
FAND1-2                   ?0??>
FAND1-3                   ??0?>
FAND1-4                   ???0>
```

```
PRIMITIVE=AND5
INPUT 5 : 1=A,2=B,3=C,4=D,5=E
OUTPUT 1 : 6=FAND
DELAY 1
FAULTLIST 11 :
A1,FAND1                    01111>
B1,FAND1                    10111>
C1,FAND1                    11011>
D1,FAND1                    11101>
E1,FAND1                    11110>
A0,B0,C0,D0,E0,FAND0        11111<
FAND1-1                     0????>
FAND1-2                     ?0???>
FAND1-3                     ??0??>
FAND1-4                     ???0?0
FAND1-5                     ????0>

PRIMITIVE=NAND2
INPUT 2 : 1=A,2=B
OUTPUT 1 : 3=FNAND
DELAY 1
FAULTLIST 4 :
A0,B0,FNAND1     11>
A1,FNAND0        01<
B1,FNAND0        10<
A1,B1,FNAND0     00<

PRIMITIVE=NAND3
INPUT 3 : 1=A,2=B,3=C
OUTPUT 1 : 4=FNAND
DELAY 1
FAULTLIST 8 :
A0,B0,C0,FNAND1 111>
A1,FNAND0        011<
B1,FNAND0        101<
C1,FNAND0        110<
A1,B1,C1,FNAND0 000<
FAND0-1          0??<
FAND0-2          ?0?<
FAND0-3          ??0<

PRIMITIVE=NAND4
INPUT 4 : 1=A,2=B,3=C,4=D
OUTPUT 1 : 5=E
DELAY 1
FAULTLIST 10 :
A0,B0,C0,D0,FNAND1       1111>
A1,FNAND0                0111<
B1,FNAND0                1011<
```

```
C1,FNAND0                1101<
D1,FNAND0                1110<
A1,B1,C1,D1,FNAND0       0000<
FNAND0-1                 0???<
FNAND0-2                 ?0??<
FNAND0-3                 ??0?<
FNAND0-4                 ???0<

PRIMITIVE=OR2
INPUT 2 : 1=A,2=B
OUTPUT 1 : 3=C
DELAY 1
FAULTLIST 4 :
A1,B1,C1 00>
A0,C0    10<
B0,C0    01<
C0       11<

PRIMITIVE=OR3
INPUT 3 : 1=A,2=B,3=C
OUTPUT 1 : 4=FOR
DELAY 1
FAULTLIST 6 :
A1,B1,C1,FOR1   000>
A0,FOR0         100<
B0,FOR0         010<
C0,FOR0         001<
FOR0            110<
FOR0            111<

PRIMITIVE=NOR2
INPUT 2 : 1=A,2=B
OUTPUT 1 : 3=FNOR
DELAY 1
FAULTLIST 4 :
A1,B1,FNOR0     00<
B0,FNOR1        01>
A0,FNOR1        10>
FNOR1           11>

PRIMITIVE=NOR3
INPUT 3 : 1=A,2=B,3=C
OUTPUT 1 : 4=FNOR
DELAY 1
FAULTLIST 7 :
A1,B1,C1,FNOR0  000<
A0,FNOR1        100>
B0,FNOR1        010>
C0,FNOR1        001>
```

```
FNOR1-1          1??>
FNOR1-2          ?1?>
FNOR1-3          ??1>

PRIMITIVE=NOR4
INPUT 4 : 1=A,2=B,3=C,4=D
OUTPUT 1 : 5=FNOR
DELAY 1
FAULTLIST 9 :
A1,B1,C1,D1,FNOR0        0000<
A0,FNOR1                 1000>
B0,FNOR1                 0100>
C0,FNOR1                 0010>
D0,FNOR1                 0001>
FNOR1-1                  1???>
FNOR1-2                  ?1??>
FNOR1-3                  ??1?>
FNOR1-4                  ???1>

PRIMITIVE=MULT44
INPUT 8 : 1=A3,2=A2,3=A1,4=A0,5=B3,6=B2,7=B1,8=B0
OUTPUT 8 : 9=M7,10=M6,11=M5,12=M4,13=M3,14=M2,15=M1,16=M0
DELAY 1

PRIMITIVE=A554
INPUT 10 : 1=A4,2=A3,3=A2,4=A1,5=A0,6=B4,7=B3,8=B2,9=B1,10=B0
OUTPUT 4 : 11=S3,12=S2,13=S1,14=S0
DELAY 1

PRIMITIVE=A22225
INPUT 8 : 1=A3,2=A2,3=A1,4=A0,5=B3,6=B2,7=B1,8=B0
OUTPUT 5 : 9=S4,10=S3,11=S2,12=S1,13=S0
DELAY 1

PRIMITIVE=A22235
INPUT 9 : 1=A3,2=A2,3=A1,4=A0,5=B3,6=B2,7=B1,8=B0,9=C0
OUTPUT 5 : 10=S4,11=S3,12=S2,13=S1,14=S0
DELAY 1

END
```

APPENDIX C. SAMPLE MAIN PROGRAM

This is a sample main program to illustrate how the simulator is called. The simulator is designed as a set of callable subroutines, some routines which perform IO are optional. The simulator was designed this way to allow the user to code special purpose main programs to suit the IO needs for each system to be simulated. The main program can be coded to interactively interrogate the user for input and simulator options in a form suitable to the user, and then restructure the data in the form the simulator expects. This design also allows the simulator to be called from other programs such as test pattern generators or other simulators.

```
#include <stdio.h>
#include "../lib/struct.h"       /* simulator type definitions */
#include "../lib/mainglobals.h" /* global variable declarations */

main() {
  int initsim(), setobserve(), evaluate(), displaylists();
  int i, length, level, modtype, activity;
  char inline[256];
  struct sigvector vector;

  /* This routine reads the fault library and system descriptions      */
  /* then links the two with the functional procedures.                */
  initsim();

  /* This routine interrogates for the pin numbers the user wishes     */
  /* to observe fault behavior.  It flags these pins so the            */
  /* displaylists routine will output faults propagated to these pins. */
  /* This routine is optional vectorflags is used only by displaylists */
  setobserve(vectorflags);

  while (1) {  /* interactive IO routine */
    for (i=0; i<=MAXSIGS; i++) { /*  Init the signal vector */
```

```
      vector.value[i]=NULL;
      vector.faultlist[i]=NULL;
    }
    fprintf(stderr,"Olease input a vector:  ");
    if ((length = getline(inline,MAXSIGS)) == 0) break;
    for (i=1; i<=length; i++) vector.value[i] = inline[i-1];
    level = 0;   /* flag the outer level for evaluate */
    modtype = 2; /* the outer level must be a scald module     */
    /* call to the evaluator, level and modtype should be 0,2  */
    /* vector contains the input vector, activity is unused    */
    /* scaldroot is a pointer set by the initialization        */
    evaluate(scaldroot,&vector,modtype,level,&activity);

    /* This routine processes faults attached to the output vector */
    /* Only the pins flagged in vectorflags are considered.        */
    /* The reporting is done in terms of the system hierarchy.     */
    displaylists(&vector,vectorflags);
  }
  fprintf(stderr,"Good-Bye...0);
}
```

REFERENCES

[1] T. W. Williams and K. P. Parker, "Design for Testability - A Survey," Proceedings of the IEEE, vol. 71, pp. 98-112, January 1983.

[2] M. Feuer, "VLSI Design Automation: An Introduction," Proceedings of the IEEE, vol. 71, pp. 5-9, January 1983.

[3] A. R. Newton, "Techniques for the Simulation of Large-Scale Integrated Circuits," IEEE Transactions on Circuits and Systems, vol. CAS-26, pp. 741-749, September 1979.

[4] V. D. Agrawal, A. K. Bose, P. Kozak, H. N. Nham, and E. Pacas-Skewes, "A Mixed-Mode Simulator," IEEE Design Automation Conference, pp. 618-625, 1980.

[5] C. H. Sequin, "Managing VLSI Complexity: An Outlook," Proceedings of the IEEE, vol. 71, pp. 149-166, January 1983.

[6] A. Yamada, N. Wakatsuki, and S. Funatsu, "Designing Digital Circuits with Easily Testable Consideration," IEEE Test Conference, pp. 98-102, November 1978.

[7] P. S. Bottorff, R. E. France, N. H. Garges, and E. J. Orosz, "Test Generation for Large Logic Networks," IEEE Design Automation Conference, pp. 479-485, June 1977.

[8] M. A. Breuer and A. D. Friedman, Diagnosis and Reliable Design of Digital Systems. Woodland Hills, California: Computer Science Press, 1976.

[9] D. B. Armstrong, "A Deductive Method for Simulating Faults in Logic Circuits," IEEE Transactions on Computers, vol. C-21, pp. 464-471, May 1972.

[10] E. G. Ulrich and T. Baker, "Concurrent Simulation of Nearly Identical Digital Networks," Computer, vol. 7, pp. 39-44, April 1974.

[11] H. Y. Chang, S. G. Chappell, C. H. Elmendorf, and L. D. Schmidt, "Comparison of Parallel and Deductive Fault Simulation Methods," IEEE Transactions on Computers, vol. C-23, pp. 1132-1138, November 1974.

[12] M. Abramovici, "A Hierarchical, Path-Oriented Approach to Fault Diagnosis in Modular Combinational Circuits," IEEE Transactions on Computers, vol. C-31, pp. 672-677, July 1982.

[13] M. R. Genesereth, "Diagnosis Using Hierarchical Design Models," HPP-81-20, Stanford University, Stanford, California, 1981.

[14] B. T. Preas and C. W. Gwyn, "General Hierarchical Automatic Layout of Custom VLSI Circuit Masks," Design Automation & Fault-Tolerant Computing, vol. 3, pp. 41-48, 1979.

[15] C. Niessen, "Hierarchical Design Methodologies and Tools for VLSI Chips," Proceedings of the IEEE, vol. 71, pp. 66-75, January 1983.

[16] P. N. Yianilos, "A Dedicated Comparator Matches Symbol Strings Fast and Intelligently," Electronics, vol. 56, pp. 113-117, December 1983.

[17] N. Giambiasi, A. Miara, and D. Muriach, "Methods For Generalized Deductive Fault Simulation," IEEE Design Automation Conference, pp. 386-393, 1980.

[18] Y. H. Levendel and P. R. Menon, "Fault-Simulation Methods - Extensions and Comparison," Bell System Technical Journal, vol. 60, pp. 2235-2259, November 1981.

[19] E. Ulrich, D. Lacy, N. Phillips, J. Tellier, M. Kearney, T. Elkind, and R. Beaven, "High-Speed Concurrent Fault Simulation with Vectors and Scalars," IEEE Design Automation Conference, pp. 374-380, 1980.

[20] M. E. Lesk and E. Schmidt, "Lex - A Lexical Analyzer Generator," in UNIX Programmer's Manual. Murray Hill, New Jersey: Bell Laboratories, 1979.

[21] S. C. Johnson, "Yacc: Yet Another Compiler-Compiler," in UNIX Programmer's Manual. Murray Hill, New Jersey: Bell Laboratories, 1979.

[22] J. D. Lesser and J. J. Shedletsky, "An Experimental Delay Test Generator for LSI Logic," IEEE Transactions on Computers, vol. C-29, pp. 235-248, March 1980.

[23] V. V. Nickel, "VLSI - The Inadequacy of the Stuck-At Fault Model," IEEE Test Conference, pp. 378-381, 1980 .

[24] C. Liaw, S. Y. H. Su, and Y. K. Malaiya, "Test Generation for Delay Faults Using Stuck-At-Fault Test Set," IEEE Test Conference, pp. 167-175, 1980.

[25] D. R. Schertz and G. Metze, "A New Representation for Faults in Combinational Digital Circuits," IEEE Transactions on Computers, vol. C-21, pp. 858-866, August 1972.

[26] E. J. McCluskey and F. W. Clegg, "Fault Equivalence in Combinational Logic Networks," IEEE Transactions on Computers, vol. C-20, pp. 1286-1293, November 1971.

[27] T. M. McWilliams, J. B. Rubin, L. C. Widdoes, and S. Correl, SCALD II User's Manual. Lawrence Livermore Laboratory, 1979, Annual Report, The S-1 Project.

[28] B. Salefski, "D," Masters Thesis, University of Illinois, Urbana-Champaign, 1981.

[29] J. P. Hayes, Computer Architecture and Organization. New York, N. Y.: McGraw-Hill, 1978, pp. 180-190.

[30] UNIX Programmers Manual. Murray Hill, New Jersey: Bell Laboratories, 1980.

END

FILMED

3-85

DTIC